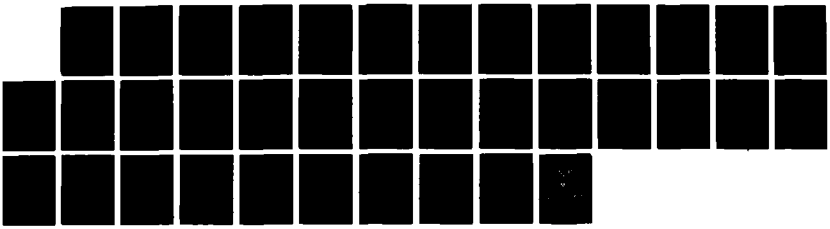
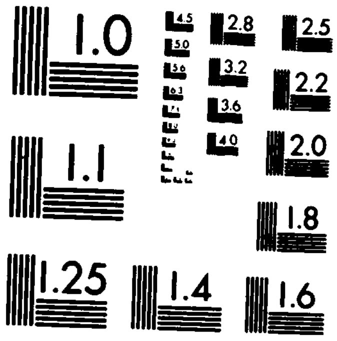
AD-A188 161   EQL: THE LANGUAGE AND ITS IMPLEMENTATION(U) NORTH CAROLINA UNIV AT CHAPEL HILL DEPT OF COMPUTER SCIENCE B JAYARAMAN ET AL. 1986 N00014-86-K-0680   1/1

UNCLASSIFIED   F/G 12/5   NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

EqL: the language and its implementation[†]

Bharat Jayaraman
Gopal Gupta

Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27514

bstract

EqL is a general-purpose language that combines the capabilities of functional and logic programming languages. A program in EqL consists of a collection of conditional, pattern-directed rules, where the conditions are expressed as a conjunction of equations, and the patterns are terms built up of data-constructors and basic values. The computational paradigm in EqL is equation solving. ~~In~~ this paper ~~we~~ describe EqL informally, by first presenting several examples illustrating the various features of the language: nondeterminism, logical variables, deferred evaluation of primitives, and user-defined constructors. This paper also describes the novel aspects of a sequential implementation for EqL: compile-time flattening and reordering of equations; and run-time equation-delaying, last-equation optimization, and rule-indexing.

† This research is supported by grant NSF-DCR-8603609 from the National Science Foundation and contract N 00014-86-K-0680 from the Office of Naval Research.

I. Introduction

Amongst a number of recent proposals for integrating functional and logic programming [RS82, D83, F85, GM84, DP85, YS86], one promising approach is *equational programming* [F85, DP85, YS86]. In our recent formulation of equational programming [JS86, JG87, J87], a program is a collection of *conditional rewrite rules*, where the conditions are expressed as a set of *equations*, and the top-level goal to be solved is also a set of equations. We have shown that this framework has the capabilities of first-order functional and Horn logic programming, with the following properties:

- pattern-directed rules;
- functional, rather than relational, notation;
- declarative semantics based on *complete set of solutions*;
- operational semantics based on *object refinement*;
- potential for parallel execution.

The computational paradigm underlying our class of equational languages is *equation solution*. This paradigm subsumes expression evaluation in a functional language because an expression e to be evaluated is simply viewed as an equation $v = e$, where v is some distinct variable. It also subsumes goal solution in logic languages because a goal g to be solved can also be viewed as an equation, namely, $g = true$. Because solutions are not necessarily unique, the declarative semantics for equations is expressed as the *complete set of solutions*. Our semantics differs from related approaches in that it is *domain-theoretic*, rather than based on *equational logic* [F85, O85, YS86]. Thus constructors and function symbols are sharply distinguished in our language, and the denotation of an expression is a set of *terms* over the constructors. *Object refinement* is a *sound* and *complete* operational strategy for computing solutions that are at least as general as the complete set of solutions [JS86].

In this paper, we describe a concrete realization of our equational programming paradigm via a language called EqL, for Equational Language. We illustrate with examples the major capabilities of EqL: nondeterminism, logical variables, deferred evaluation of primitives, and user-defined constructors. We also describe the salient features of a sequential implementation of EqL, written in C. The current implementation has two phases:

1. *compilation*, in which the source code is transformed by *flattening* and *reordering* equations, in accordance with their semantics; and

2. *interpretation*, in which equations are solved via a depth-first backtracking search using a five-stack execution model. The novel aspects of this model are (i) the separation of

the conventional recursion stack into two parts: the *control* stack and the *variable* stack, and (ii) the use of *equation-delay* and *equation-trail* stacks for deferred (but not lazy [HM76]) evaluation of certain primitives. In addition, a *trail* stack is used for recording which variables must be undone upon backtracking, as in Prolog implementations [WPP77].

Data objects are stored in a *heap*, as in conventional language implementations. The data representation is based on Boyer and Moore's *structure sharing* [BM72], a technique for minimizing copying when creating structured objects. The separation of *control* information from *variable* information facilitates early reclamation of the control frame. It also enables *Last-Equation Optimization* (LEO), which is similar to Tail-Recursion Optimization (TRO) in functional language implementations. LEO actually is more general than TRO because it is applicable to certain non-tail-recursive definitions, such as LISP's append, in which the last call is embedded inside a data-constructor at the outermost level.

The rest of this paper is organized as follows: section II describes the data objects of EqL and the informal meaning of EqL rules. Section III briefly describes interaction with the EqL interpreter, and section IV presents examples of EqL programs for functional and logic programming. Sections IV and V describe respectively the compilation and interpretation interpreter of EqL. Section VI presents conclusions and further comparisons with related work.

II. EqL Language Features

In this section we describe the data objects in EqL and the informal meaning of rules.

II.1. Data Objects

The data objects in EqL are defined below:

(i) *Numbers*: The current implementation of EqL provides only integers, e.g. `10, 207, 0, −11, −3999,` etc.

(ii) *Booleans*: `true, false`.

(iii) *Atoms*: Any identifier beginning with an upper-case letter or any sequence of characters enclosed within single quotes, e.g. `Apple, EqL, 'also an atom'`, etc.

(iv) *Variables*: Normally begin with a lower-case letter, e.g. `x, y, tree, p1, q1,` etc. "Anonymous" variables begin with the underscore symbol, and serve as place holders in data structures, e.g. `cons(h, _dontcare)`. The underscore symbol by itself is also an anonymous variable.

(v) *Structures*: In the current implementation of EqL, there are two built-in structured data objects: *trees* and *strings*. User-defined structures may be specified using the `constructor` declaration, explained later. We explain trees and strings below.

We use the word *term* to refer to any data object of EqL that is built up from the above entities. We sometimes use the word *structured term* to refer to a term that has a constructor at the outermost level.

As in LISP, the built-in constructor `cons(x,y)` defines a binary tree, and the operations `car(t)` and `cdr(t)` access the left- and right-subtree of a tree `t` respectively. Examples:

```
cons(10, 20),
cons(10, cons(20,30)),
cons(cons(a,10), cons(B, 20))
```

In the third example, note that a is a variable, but B is a atom.

Because list-processing is a common application of functional and logic languages, EqL provides a special notation for *lists*. Similar to lists in LISP and Prolog [CM81], EqL lists are a special case of trees; they correspond to trees which "slope to the right" and end with the special symbol `[]`. The following examples illustrate the connection between lists and trees:

List Notation	*Tree Notation*
`[1,2,3]`	`cons(1,cons(2,cons(3,[])))`
`[[[1]]]`	`cons(cons(cons(1,[]),[]),[])`
`[]`	`[]`

Similar to PROLOG, the notation

```
[ x | y ]
```

is used to stand for `cons(x,y)`.

String constants are defined by a sequence of characters enclosed within a pair of double-quotes, e.g. `"abc"`, `"123"`, `"longer string"`, etc. The empty string is `""`. Analogous to the above notation for lists, we use

```
[ x :  y ]
```

to refer to the string obtained by prepending the one-character atom denoted by variable x in front of string denoted by y. For example, `['a' :  "bc"]` is the string `"abc"`, and `['a' :  ""]` is the string `"a"`.

II.2. Rules

We illustrate program rules through examples. Below is a program to find the maximum depth of a binary tree of integers.

```
depth(x) => 0 where numberp(x) = true
depth([left | right]) => if dl > dr then dl+1 else dr+1
                             where
                                         dl = depth(left);
                                         dr = depth(right).
```

Program II.1: Maximum depth of a binary tree

The above program has two rules, which define the two cases for the depth function: the case for a leaf (an integer), and the case for a nonleaf (a tree built up from cons). In general, an EqL rule may take one of two forms:

1. *f(patterns)* => *expression.*
2. *f(patterns)* => *expression* where *equations.*

We explain each component of a rule below:

(i) *Patterns.* The word *pattern* is a synonym for *term*, which was defined in the previous section. Two or more patterns in a sequence are separated by commas, e.g.,

```
f([x1 | y1], [x2 :  y2], z) => ...
```

Zero-argument operations are permitted, and are defined by

```
f() => ...
```

(ii) *Expressions.* Every primitive or user-defined operation in EqL, except if-then-else, is *strict* in all of its arguments, i.e., its result is undefined if any of its arguments is undefined. The if-then-else is strict only in its first argument. The different kinds of expressions in the current implementation are listed below:

a. Terms: Every term is also an expression.

b. Primitive Operators: There are three kinds of primitive operators: arithmetic, relational, and boolean. These operators are summarized below:

Arithmetic: `+`, `-`, `*`, `div`, `mod`, and unary `-`. The binary operators + and - have lower precedence than * and `div`, which in turn have lower precedence than `div` and `mod`. Unary - has the highest precedence. The operator `div` returns the integer quotient. All binary operators are left associative. In addition, the function `abs(x)` returns the absolute value of x.

Relational: `<`, `>`, `<=`, and `>=`. The equality symbol = is reserved for defining equations. The function `eq(x,y)` maps identical atoms, numbers, and booleans to `true`; otherwise it returns `false`. The functions `lessp(x,y)`, `greaterp(x,y)`, `lesseq(x,y)`, and `greatereq(x,y)` may be used in place of `<`, `>`, `<=`, and `>=` respectively.

Boolean: `and`, `or` and `not`. The operator `not` has precedence over `and`, which has precedence over `or`. The following domain predicates are provided in the language.

`numberp(x)`	: `true` if x is a number; `false` otherwise.
`boolean(x)`	: `true` if x is a boolean; `false` otherwise.
`atom(x)`	: `numberp(x) or boolean(x)`.
`null(x)`	: `eq(x,[])`.

c. If then else:

`if` *boolean-expr* `then` *expr1* `else` *expr2*

Both the `then`- and the `else`-part may have equations associated with them. Thus, for example,

(*expr1* `where` *equations*)

may be used in place of *expr1* above. We explain *equations* further below.

d. Application:

`f`(*arguments*)

where `f` is the name of some user-defined function and *arguments* is a sequence of zero or more expressions separated by commas.

EqL also supports I/O operations, but we do not discuss them in this paper. A complete description of I/O operations may be found in [JG87].

(iii) *Equations*. An equation is of the form

$$term = expression \qquad \text{or} \qquad expression = term.$$

Although permitted by our implementation, an equation $expression_1 = expression_2$ offers no extra power since it is equivalent to the pair of equations: `v`= $expression_1$; `v` = $expression_2$, where `v` is a distinct variable. Actually, it suffices to permit equations of the form `v` = *expression*, but permitting a *term* in place of variable `v` often leads to fewer intermediate variables in the source program, and hence clearer definitions.

Two or more equations in a sequence are separated by semi-colons. Because the interpreter is sequential, it solves equations in the sequence presented. In the equation-solving rules described informally below, we refer to expressions being 'evaluated', but the reader should note that an expression e is evaluated by solving an equation $v = e$. We assume henceforth that an equation is of the form *term* = *expression*; the symmetric case is treated similarly. There are four cases for an equation, corresponding to the four forms of an *expression*:

1. $term_1 = term_2$. This is solved by syntactic unification. Failure to unify initiates backtracking.

2. *term* = $\mathtt{f}(e_1, \ldots, e_n)$, where each e_i is an *expression*, and `f` is a user-defined operation, whose first rule is

   $\mathtt{f}(t_1, \ldots, t_n)$ `=>` *expression* `where` *equations*.

   The equation reduces to the following sequence of equations, where each v_i is a distinct variable:

   $v_1 = e_1; \ldots; v_n = e_n; \quad v_1 = t_1; \ldots; v_n = t_n; \quad$ *equations*; $\quad$ *term* = *expression*.

   That is, the expressions $e_1 \ldots e_n$ are first evaluated and their results then unified with $t_1 \ldots t_n$ respectively; the *equations* in the body of `f` are then solved; and finally, the *expression* in the body of `f` is evaluated and unified with *term* similarly. If, at any stage, unification fails backtracking occurs to the dynamically most recent operation having an untried rule. Note that the evaluation order is essentially *call-by-value*. Also, the implementation ensures that ther computed value for each e_i above is not re-computed when an alternative rule for `f` is attempted upon backtracking.

3. *term* = `if` p `then` e_1 `else` e_2. If expression p evaluates to a boolean, the equation *term* = e_1 or *term* = e_2 is solved. If expression p evaluates to a number or structured term, the equation fails and initiates backtracking. If p evaluates to an unbound variable, say x, two nondeterministic paths arise: (1) the equation *term* = e_1 is to be solved, where x $\leftarrow$ `true`; and (2) the equation *term* = e_2 is to be solved, where x $\leftarrow$ `false`. These two paths are tried out sequentially, with backtracking. Note that if e_1 or e_2 were accompanied by equations, these equations would be solved *before* solving *term* = e_1 or *term* = e_2.

4. *term* = *primitive*. The arguments of the primitive operator are first evaluated. If their results are not of the proper type, the equation fails and initiates backtracking; otherwise, the operator is applied to the arguments to produce a result, which is then unified with *term*. If the arguments reduce to unbound variables, the equation is *deferred*, or *delayed*, until all unbound variables are fully defined.

All these cases are clarified through examples in the next two sections.

III. The EqL Interpreter

In this section, we give a brief account of the EqL interpreter implemented under Unix. More details may be obtained from [JG87].

The top-level query of an EqL program is either an expression or an equation or a set of equations, terminated by a period. A top-level expression e is actually treated internally as an equation, _ = e, where _ is the anonymous variable.

A typical session in EqL is a "conversation" between the user and the interpreter. The interpreter is first invoked by the command

```
% eql
```

where we assume % is the Unix command-level prompt. The interpreter would respond as follows:

```
EqL Version 1.0

eql>
```

Often, a set of EqL rules are kept in a file, which can be read in by a consult operation (as in C-Prolog). For example, if depth is the name of a Unix file containing the two rules for the depth function, it can be read in as follows.

```
eql> consult('depth').
```

EqL will respond with

```
true

eql>
```

Now, the depth function can be invoked on some input tree, e.g., [10 | [20 | [30 | 40]]], as follows:

```
eql> depth([10 | [20 | [30 | 40]]]).
```

The interpreter would respond with

```
3

eql>
```

Note that the top-level query can, in general, be a sequence of equations. For example, the following goal is equivalent to depth([10 | [20 | [30 | 40]]]):

```
eql> x = depth([10 | [20 | [30 | 40]]]).
```

The interpreter's response to this goal would be:

```
x = 3

eql>
```

Actually, the top-level query depth([10 | [20 | [30 | 40]]]) would in fact be internally converted into an equation, _ = depth([10 | [20 | [30 | 40]]]).

If the EqL interpreter finds that it cannot solve the top-level query, it would respond with the message,

```
no solution
```

This might happen, for example, when the top-level query is:

```
eql> depth([]).
```

because [] is not an atom. To account for this case, an explicit rule,

```
depth([]) => 0.
```

can be provided.

III.2. Obtaining Multiple Solutions

Consider the following definition for the familiar append function of LISP, for non-destructively concatenating two lists:

```
append([], x) => x.
append([h|t], y) => [h | append(t,y)].
```

Program III.1: List append

For example, the result of the query

```
eql> append([1, 2], [3, 4]).
```

would be the list

```
[1, 2, 3, 4].
```

It is just as easy to find out the lists x and y such that when appended together will yield the list [1,2,3,4]. This query can be expressed as follows:

```
eql> append(x,y) = [1,2,3,4].
```

The interpreter will respond with:

```
x = []
y = [1, 2, 3, 4]
```

Upon typing a semi-colon at the end of the second line above, the interpreter will respond with the second solution:

```
x = [1]
y = [2, 3, 4]
```

Typing a carriage return instead of a semi-colon will cause the interpreter to discard remaining solutions and return with the

```
eql>
```

prompt. All five solutions to the above query can be inspected by typing a semi-colon at the end of each preceding solution.

Thus, the interpreter for EqL, like a Prolog interpreter, explores alternative solutions to a query by depth-first search with backtracking.

IV. Programming in EqL

We now illustrate through examples the various features of EqL: non-determinism, delayed evaluation of primitives, logical variables, and user-defined constructors.

IV.1. Nondeterminism

We already saw in the previous section that multiple solutions may exist for an equation. We illustrate nondeterminism in EqL further through two examples: the family database, and the N Queens problem.

IV.1.1. Family Database

Shown below are four operations: f(x), which returns the father of some person x; m(x), which returns the mother of x; p(x), which returns the parent of x; and gp(x), which returns the grand-parent of x.

```
f(Bob)   => Gary.
m(Bob)   => Mary.
f(Ann)   => Gary.
m(Ann)   => Mary.
f(Gary)  => Joe.
m(Gary)  => Jane.
f(Mary)  => Steve.
m(Mary)  => Sue.
p(x)     => f(x).
p(x)     => m(x).
gp(x)    => p(p(x)).
```

Program IV.1 : Family Relationships

The operation p(x) is nondeterministic because there are two rules for operation p. This reflects the fact that a person in this database has more than one parent—two to be precise. Because gp(x) is defined in terms of p(x), it is easy to see that gp(x) is also nondeterministic. The grand-parent of some person, say Bob, could be found by:

```
eql> gp(Bob).
```

The first answer produced by the interpreter would be Joe, because p(Bob) would first return Gary, and p(Gary) would first return Joe. Upon requesting the next answer (by typing semi-colon), the interpreter would backtrack to the latest point where another choice is possible. Thus p(Gary) is recomputed, via m(x), to be Jane, which becomes the next answer to the top-level query. The other answers, Steve and Sue, are determined similarly.

The grand-children of some person, say Joe, could be found with query such as:

```
    eql> gp(x) = Joe.
```

IV.1.2. N Queens Problem

We now present a more complex example of nondeterministic programming. The problem is to place N queens on an N by N chess board in such a way that no two queens are attacking one another. A simple approach to this problem is to place queens on successive columns so that each new queen placed is not attacked by any queen in the preceding columns. If a queen cannot be placed on a given column, we go back to the preceding column to see if the queen there has another "safe" position. If there are no safe positions remaining on that column, we back up to the preceding column, and so on. A solution is found if we can thus place queens on all N columns. We have exhausted all solutions if we attempt to go back from the first column to the 0-th column.

Below is an EqL program for specifying the desired search:

```
queens(n) => solve (1, [], n).
solve(col, safelist, n) => if eq(col,n+1) then safelist
                           else place ([col | row(n)], safelist, n).
place(q, safelist, n) => solve (col+1, [q|safelist], n)
                                 where q = [col|row];
                                       safe(safelist, q) = true.
safe([],q) => true.
safe([q1 | t], q) => safe(t,q) where threatened(q, q1) = false.
threatened([c1|r1], [c2|r2]) => eq(r1,r2) or eq(abs(r1-r2), abs(c1-c2)).
row(n) => n              where greaterp(n, 0) = true.
row(n) => row(n-1)       where greaterp(n, 0) = true.
```

Program IV.2 : N Queens Problem

The nondeterminism in the above program lies in the operation `row(n)`, which generates the sequence of integers n, n-1, ..., 1, one at a time. This provides a way for stepping through the rows in any particular column. This example also shows the clarity of function composition and `if then else`; a comparable Prolog definition must rely on the "cut" and other extra-logical features for arithmetic to specify the desired solution.

IV.2. Delayed Evaluation of Primitives

An interesting aspect of the execution of an EqL program is the way primitive operators are handled. Basically, when a primitive operation, such as `+`, `atom(x)`, `>`, etc., has all of its arguments defined, it is simplified to produce a result. However, when its arguments are not sufficiently defined when first encountered, it will be deferred until sufficient information is available to simplify it. This delayed evaluation has many uses, as we will illustrate in this section. A similar form of delaying has also been recently used in the CLP language [JL87].

IV.2.1. Simulating Sets with Lists

The following rules define the familiar LISP operation member, which tests if an element x is a member of a list.

```
member([],    x) => false.
member([x|t],x) => true.
member([y|t],x) => member(t,x) where eq(x,y) = false.
```

Program IV.3 : List Membership

It is easy to see that member will correctly check if an element, say 3, is a member of some list, say, [1,2,3,4,5]. Member could just as easily be used to enumerate the elements of a list, using a goal such as

```
eql> member([1,2,3,4,5], z) = true.
```

which would return 1, 2, 3, 4, and 5 as the value for z, one at a time. The goal below, however, illustrates a different outcome:

```
eql> member([1,2,3,4,5], z) = false.
```

Here, the first rule of member fails because [] does not match [1,2,3,4,5]. The second rule initially succeeds in unifying the goal arguments with its patterns, but its result, true, fails to match false in the top-level query. Thus the third rule of member is taken. When the operation eq(x,y) is encountered, note that y has the value 1 but x is unbound. The EqL interpreter therefore defers this equation (by moving it to a global stack of such deferred equations), and proceeds with the recursive call on member. Each succeeding call on member results in one new deferred equation, eq(z,2) = false, eq(z, 3) = false, etc. Finally, the first argument to member is [], and the first rule succeeds, and all equations are solved, except for the deferred equations. Because z is still unbound, the EqL interpreter responds as follows:

```
Input equations not fully constrained
z = _16
eql>
```

The binding of z to a number preceded by the under-score symbol indicates that z is unbound. Although the above behavior might not seem useful at first, consider the following natural definition of set difference (in terms of member).

```
diff(x,y) => d where member(y, d) = false;
                     member(x, d) = true.
```

Program IV.4 : Set Difference

The above rule states that the difference of two sets x and y (represented as lists) consists of elements d such that d is not a member of y and d is member of x. For example, the goal

```
eql> diff([1,2,3,4,5], [1,3,5,6,7]).
```

will return 2 and 4 as answers, one at a time.

EqL is able to find these answers as follows. At the end of solving the first equation, `member(y, d) = false`, where y = `[1,3,5,6,7]`, there will be five deferred equations:

```
eq(d,1) = false; eq(d,3) = false; ...  ; eq(d,7) = false.
```

When attempting solve `member(x, d) = true`, with x = `[1,2,3,4,5]`, the binding of d to the values `1, 3, or 5` will cause one of the deferred equations to fail, and hence these are determined not to be solutions. However, the binding of d to the values 2 or 4 will cause all the deferred equations to succeed, and hence these are determined to be solutions.

EqL's support for inequality represents one of its strengths over Prolog, which cannot define `member` or `diff` in this generality. Because of deferred evaluation, the order of the two equations in the definition of `diff` does not affect the correctness of the program.

IV.2.2. Arithmetic primitives

Deferring the evaluation of arithmetic primitives also has some interesting uses. Consider the conversion of centigrade to fahrenheit, expressed by the following rule:

```
f(c) => 32 + 9*c div 5.
```

This rule can be used to convert centigrade to fahrenheit by a goal such as

```
eql> f(100).
```

It can also be used to find the centigrade for some particular fahrenheit, by a goal such as

```
eql> f(x) = 212.
```

To understand the process by which this equation is solved, it should noted that EqL converts the above rule into the following program:

```
f(c) => 32 + t2 where t1 = 9*c; t2 = t1 div 5.
```

The top-level goal, `f(x) = 212`, results the following sequence of equations to be solved:

```
t1 = 9*x; t2 = t1 div 5; 32 + t2 = 212.
```

EqL defers the first two equations, and solves the last equation to obtain the value of t2. With this information t1 is determined from the second equation, and finally x from the first equation. Note that EqL will solve equations of the form $c = a\ op\ b$ where *op* is an arithmetic operator (+, -, *, div) and two of the three arguments *(a, b, c)* are known. With this capability, the reader may verify that the interpreter will be able to find the x such that

```
eql> fact(x) = 24.
```

where `fact` is the familiar factorial function, defined as:

```
fact(0) => 1.
fact(n) => n*fact(n-1) where greaterp(n,0) = true.
```

We also leave it to the reader to explain why, for example, `fact(x) = 23` will fail to terminate.

IV.3. Logical Variables

Much of the power in logic programming lies in its "logical variables." All variables in EqL are logical variables in that they derive their values not by direct binding but by the satisfaction of constraints. A classic example of logical variables is its use in defining "difference lists", which permit list concatenation in constant-time. The following is the EqL definition of difference-list concatenation.

```
dconc([x|t], [t|y]) => [x|y].
```

Difference lists can be used to avoid the use of append in many places. Consider, for example, the following inefficient definition of quick-sort.

```
qsort ([]) => [].
qsort([p|l]) => append(qsort(a), [p|qsort(b)])
                    where [a|b] = part(l, p).
part([], p) => [[] | []].
part([h|t], p) => if p>h then [[h|a] | b] else [a | [h|b]]
                    where
                              [a|b] = part(t, p).
```

Program IV.5 : Naive Quicksort

The above program is inefficient because it employs a linear-time append operation at each stage of the sorting process. This inefficiency can be avoided by concatenating the sorted lists in constant-time by representing them as difference lists, as follows.

```
sort(l)          => answer where [answer | []] = dsort(l).
```

```
dsort([])           => [x|x].
dsort([p | l])      => [sorta | tail] where
                                   [a | b] = part(l,p);
                                   [sorta | [p| sortb]] = dsort(a);
                                   [sortb | tail] = dsort(b).
```

Program IV.6 : Quicksort with difference lists

IV.4. Strings and User-defined Structures

Consider the following definition for non-destructively concatenating two strings:

```
cat([], x) => x.
cat([h:t], y) => [h :  cat(t,y)].
```

Program IV.9 : String Concatenation

Using the above definition, we may determine the two portions, front and back, of some string s, such that they are separated by some specific word w, as follows:

```
split(w, s) => [front, back] where cat(front, cat(w, back)) = s.
```

User-defined constructors are declared before their use. They can be declared anywhere in a file between the rules, e.g.

```
constructor:  seq, fun.
```

where seq and fun are the names of the constructors. These two constructors may be used in a program for performing type inference—seq and fun stand for sequence and function respectively. For example, the following rules illustrate how these constructors may be used to define the types of some primitive functions on sequences: Nil, Head, Tail, and Adds.

```
type(Nil) => seq(x).
type(Head) => fun(seq(x), x).
type(Tail) => fun(seq(x), seq(x)).
type(Adds) => fun(x, fun(seq(x), seq(x))).
```

V. Compilation of EqL

The implementation of EqL divides into two phases:

i. *The compilation phase*: the source code is read in and transformed into a binary-tree intermediate structure.

ii. ***The interpretation phase***: the various execution stacks are set up, the query code evaluated and the results printed.

In the remainder of this section, we describe the compilation phase. Compilation involves lexical analysis, parsing and generation of intermediate structure. Lexical analysis and parsing are done using the Unix tools Lex and Yacc. Two key transformations are performed while generating the intermediate code: ***flattening*** and ***equation reordering***. Below we describe the two transformations.

V.1. Flattening

To ensure left-most inner-most order of evaluation and to facilitate the temporary storage of subexpression values prior to expression evaluation, the source code is ***flattened***. The ***flattening*** operation is simple: a subexpression e, which is not a term, is replaced by a compiler-generated variable t, and a new equation t = e is added to the existing set of equations. For example, consider the second definition of the inefficient version of qsort in section IV.3.1:

```
qsort([p|l]) => append(qsort(a), [p|qsort(b)])
                where [a|b] = part(l, p).
```

After flattening it would become:

```
qsort([p|l]) => append(t1, [p|t2])
                where [a|b] = part(l, p);
                      t1 = qsort(a);
                      t2 = qsort(b).
```

Here t1 and t2 are the compiler-generated temporary names. Note that the order of equations in flattened code reflects the desired evaluation order. As another example, the query

```
f(x) + g(h(x)) = 23.
```

after flattening becomes

```
t1 = f(x);
x1 = h(x);
t2 = g(x1);
t1 + t2 = 23.
```

where t1, t2 and x1 are compiler-generated variable names.

The reader might note that the flattened EqL code resembles the source code for the comparable Prolog definition, in that the output variables of each Prolog predicate correspond to our compiler generated names. These output variables in Prolog pose an extra burden on the programmer, who has to manage them explicitly, and also lead to less clear definitions. EqL does not bar the programmer from using such output variables, but in most cases their use can be avoided.

V.2. Equation Re-ordering

Although EqL does not specify the order of solving equations, the interpreter we have implemented solves equations sequentially. There are, however, two exceptions to this general rule, referred to as *compile-time* and *run-time* equation re-ordering. We discuss compile-time re-ordering in this section; run-time re-ordering is discussed in section VI.3.3.

In compile time re-ordering, an equation relating two *terms* is moved ahead of other kinds of equations. This re-ordering usually makes operation applications less non-deterministic, because unifying the two terms in such re-ordered equations usually results in variables being bound, and operation applications with more bound variables tend to be less non-deterministic.

Consider the query consisting of the equations

```
f(x) = term;
x = [h|t].
```

where `f` is defined by *n* rules. Solving the second equation before the first binds x to `[h|t]`. With this binding for x, unifying `f(x)` against the various rules for `f` would very likely eliminate many of them without evaluating their bodies.

V.3. Compiling if-then-else

For successful execution, the condition part of an if-then-else may reduce either to a boolean or to an unbound variable (say x). In the former case, the then-part or the else-part is evaluated depending upon the value of the boolean. In the latter case, the interpreter has two non-deterministic choices: (i) bind x to `true` and return the value of the then-part as the first solution, and (ii) bind x to `false` and return the value of the else-part as the second solution.

As an aside, the reader may note that implementing if-then-else using the rules shown below is not correct:

```
if-then-else(true, T, E) => T.
if-then-else(false,T, E) => E.
```

These rules are not a correct implementation because they cause the if-then-else to be strict in all arguments, instead of just the first argument.

In section VI.2.3 we describe the run-time support for executing an if-then-else expression. Below we illustrate the translated code for an if-then-else expression through an example. Consider the operation `solve` in the *N-queens* program.

```
solve(col, safelist, n) => if eq(col,n+1) then safelist
                           else place ([col | row(n)], safelist, n).
```

This is transformed into:

```
solve(col, safelist, n) => if bool then safelist
                            else (place ([col | t1], safelist, n)
                                      where t1 = row(n))
                               where t2 = n + 1;
                                     bool = eq(col, t2).
```

where bool is the compiler-generated name for the condition part of the if-then-else, and t1 and t2 are compiler-generated names for sub-expressions row(n) and n + 1 respectively.

V.4. Intermediate Code Structure

The intermediate code is organized as a binary tree and is stored in the *static-area* (implemented as an array of *cells*). Each node of the intermediate code tree uses exactly one cell. A typical cell has the following information:

i. **A tag** which identifies the kind of node the cell stores, *e.g.* an atom, or a primitive operator, or a rule header, etc. The tag of a node determines the operation to be performed.

ii. **A left pointer** field which stores the pointer to the left subtree for an internal node of the tree, or the actual value of the constant for a leaf node.

iii. **A right pointer** field which stores the pointer to the right subtree for an internal node of the tree, or null for a leaf node.

iv. An **info** field which stores miscellaneous information, depending on the kind of node, e.g., the hash value of a rule name for a rule header cell.

As an illustration, the intermediate code for the body of the second rule of the append function (program III.1) is shown in figure V.1.

The intermediate code of a rule is stored in the *rule store*, which is implemented as a hash table, where the hash value of a rule is the sum of the ASCII value of the characters in its name modulo 100. This value is computed and stored in the *info* field of the rule header cell at compile time. Rule definitions falling in the same bucket are stored as linear list, preserving the order they appear in the source program. Besides storing a pointer to the body of the rule, the following information is also stored:

i. rule name
ii. rule arity
iii. number of variables in the body of the rule
iv. list of formal arguments.

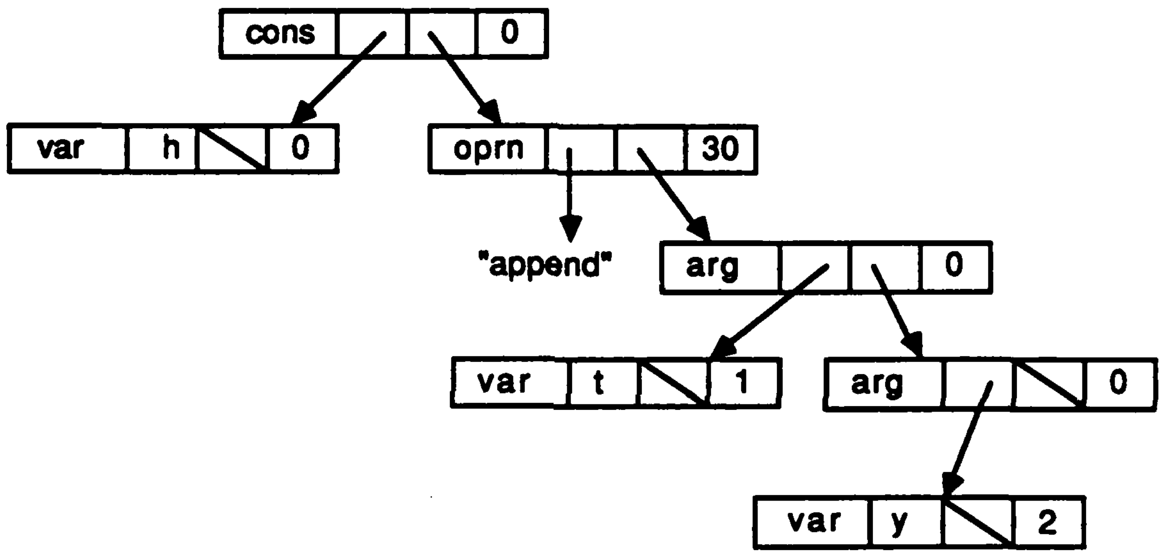


figure V.1.: Intermediate Code for append

VI. The Interpretation Phase

In this section we discuss equation solution. We also discuss the optimizations incorporated in the interpreter for more efficient execution and quantify the improvements resulting from them. Some of our terminology is borrowed from Prolog implementation descriptions in [H84] and [WPP77].

VI.1. Overview

Without loss of generality, the top level of an EqL query may be assumed to be

$$term_1 = expression_1;$$
$$\ldots$$
$$term_k = expression_k;$$

The execution process consists of solving equations from the input query sequentially. The appropriate action for solving an equation involves a case analysis of the 4 kinds of equations defined in section II:

1. $term_1 = term_2$
2. $term = f(e_1, \ldots, e_n)$
3. $term =$ `if` p `then` e_1 `else` e_2
4. $term = primitive$

The EqL interpreter solves the equations using a five-stack execution model. The five stacks are:

1. *control stack*: The control stack records the locus of control during operation applications, and is similar to the activation record stack in conventional language implementations. Each operation application or `if-then-else` evaluation results in a *frame* being pushed on the control stack.
2. *variable stack*: Space for variables in a rule and the top level query is allocated on the variable stack.
3. *trail stack*: The trail stack is used for recording the addresses of the variables whose binding is to be undone upon backtracking.
4. *equation-delay stack*: This stack is needed to implement *equation delaying*. Delayed equations are solved later when operands get bound.
5. *equation-trail stack*: The equation-trail stack is used for recording those delayed equations which have been solved, but need to be re-solved on backtracking.

Apart from these five run-time stacks the interpreter also accesses: (i) the *static area*, where the intermediate code is stored, (ii) the *I/O buffer* area, and (iii) the *molecule heap* where *molecules* are allocated—we explain *molecules* later.

Note that control and data have been separated in the execution model by having separate stacks for them. This enables reclaiming the control stack frame upon completion of an operation application (*success exit*), and also enables *last equation optimization*.

VI.2. Equation Solution

We now describe how each kind of equation is solved.

VI.2.1. Solution of $term_1 = term_2$

This kind of equation is solved by unifying $term_1$ and $term_2$. If the unification succeeds, the equation is solved; otherwise backtracking is initiated. The variables in $term_1$ and $term_2$ are allocated space on the variable stack. The unification algorithm uses a small local stack to traverse structures. The algorithm does not do an *occurs check*, so a cyclic structure may be produced during runtime. It is the programmer's responsibility to make sure that programs do not generate cyclic structures during execution.

A term is either a constant (number or boolean), a variable, a structure, or a string. Representation of constants and variables is discussed later. A structure is either a primitive *cons* or a user defined constructor. In the next subsection we describe the representation of structures, using *structure sharing* [BM72].

VI.2.1.1. Structure Sharing: Constructors and Strings

As far as the implementation is concerned, there is only one built-in basic constructor – `cons`. All other constructors defined by the user are represented in terms of this basic constructor. For example, the term `tree(left, right)` would be represented as `cons('tree', cons(left, right))` where `'tree'` is an atom. Even the data type string is implemented in terms of `cons`.

A `cons`-object is allocated using a *molecule* [WPP77]. Similar to a closure, a molecule is a pair of pointers: one to the *skeleton* code (called the *code pointer*) and the other to the *environment* of the skeleton (called the *environment pointer*). The skeleton code consists of the intermediate code residing on the static area. Thus, if we were to solve the equation `x = cons(h, t)`, where x is unbound, the variable cell for x on the variable stack would be bound to a molecule as shown in figure VI.1.

A string is implemented as list of characters, e.g., the string `"abc"` would be represented as `['a', 'b', 'c']`; the null string `" "` by the null list `[]`. A marker bit is set in each cell indicating whether it is a string cell or a list cell.

VI.2.2. Solution of $term = \mathtt{f}(e_1, \ldots, e_n)$

Solving this equation involves evaluating the operation application $\mathtt{f}(e_1, \ldots, e_n)$ and then unifying the value returned with *term*. The first step in evaluation of $\mathtt{f}(e_1, \ldots, e_n)$ is to

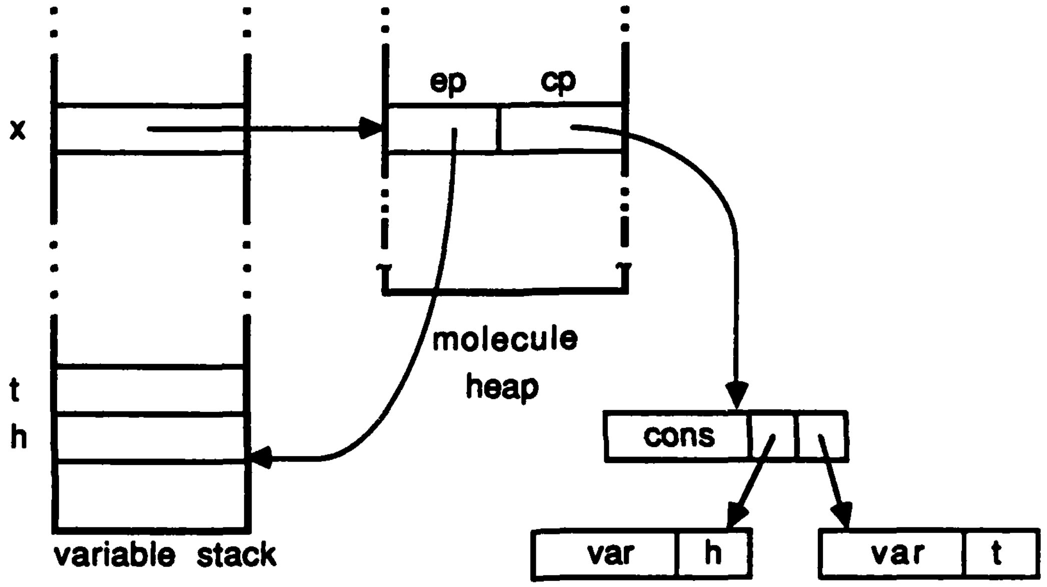


figure VI.1.: Binding a variable to a molecule obtain the definition of `f` from the *rule store*. In general, there might be more than one potential *candidate*, i.e. rules for `f` having the same arity. These candidates are tried one by one in *textual* order. Let us call these candidates C1, C2, ..., Cn, in order. The one to be tried first, C1, will have the general form:

C1: $f(t_1, \ldots, t_n)$ => *expression* where *equations*.

Suppose the actual parameters of `f` unify with the terms $t_1, \ldots, t_n$, then the invocation of C1 by `f` leads to the following equations:

equations;
term = *expression*.

If the attempt to solve these equations succeeds, the original equation has been solved, and a *success exit* is said to have been taken from C1. If, however, the attempt to solve them fails, a *failure exit* is said to have been taken, and the original equation is retried with the next candidate C2 for `f`. Note that these equations might result in more operation applications. Of course, the attempt to solve them might also be non-terminating.

In the next two subsections, we first describe the two major data structures needed in supporting operation applications: the control stack and the variable stack. We then briefly describe the control algorithm.

VI.2.2.1. Control Stack

As the interpreter solves equations, it has to remember a variety of facts about the operations applications that have taken place so far. This information is stored in a *frame* in the control stack. A typical control frame contains fields which have the following information (name of the fields given in parentheses):

i. A pointer to the code to be executed by this frame (`cp`). This code is a list of equations.

ii. A pointer to the previous backtrack point (`pb`).

iii. A pointer to the trail stack (`tp`).

iv. Next candidate to be tried on failure (`nc`).

v. A pointer to the base of the area allocated to it on the variable stack, or the environment pointer (`ep`).

vi. A return point (`rpt`).

vii. A pointer to the parent frame, i.e. the frame which created this frame (`pf`).

viii. A pointer to the equation-delay stack (`dsp`).

A frame corresponding to an operation with multiple definitions is called a *choice point*. The `tp`, `nc` and `pb` fields of a frame which is not a choice point are *nil*.

Figure VI.2 shows the state of control stack when the frame for the candidate C1, from the example in the previous section, is pushed. In the figure the *current call* register (cc) points to the frame currently executing. The last field (dsp) has been omitted in the figure.

VI.2.2.2. The Variable Stack

The local variables of an operation application are allocated space on the variable stack. A pointer to the base of this area is stored in the ep field of the operation's frame. Each cell of the variable stack has two fields: one, to store the type of the datum, and the other, to store its actual value. For atoms, numbers and booleans the datum field contains the actual value of the constant. A variable-to-variable assignment, e.g., in the equation x = w, is recorded by storing a pointer to w's cell in x's cell. Constructors (including strings) are stored using the method of *structure sharing*, as discussed earlier.

VI.2.2.3. Control Algorithm: Overview

To conclude our description of an operation application, we briefly summarize the control algorithm. The algorithm has four phases:

i. *operation invocation*: The definition of the operation is retrieved from the rule store. The definition of the next candidate of that operation is also retrieved, if there is one. A frame is created for this definition, with all the required information put in the various fields, and pushed on the control stack. The current call register (cc) is updated to point to this new frame. Space is allocated for the variables of this operation application on the variable stack, and execution is initiated.

ii. *operation execution*: The first step in the execution of the operation is the unification of its actual parameters with its formals. If the unification fails, backtracking takes place. If it succeeds, the equations in the body of the operation are solved sequentially. If any of these equations turn out to be unsatisfiable, backtracking takes place. These equations might lead to more operation applications.

iii. *operation exit*: Once all equations in the body of the operation have been solved, a *success exit* is taken from its frame. This is done by making the cc register point to the *parent* of the current frame and restarting the execution from the equation pointed to by the return point (rpt field) of the current frame. If there are no choice points above the frame pointed to by cc, all frames above it are popped, since they are no longer needed. The variable bindings made by the deleted frames are preserved on the variable stack. This is one reason for separating data and control in the interpreter.

iv. *backtracking*: When an equation turns out to be unsatisfiable, the interpreter has to backtrack and retry the most recent call with an untried rule. A pointer to the most

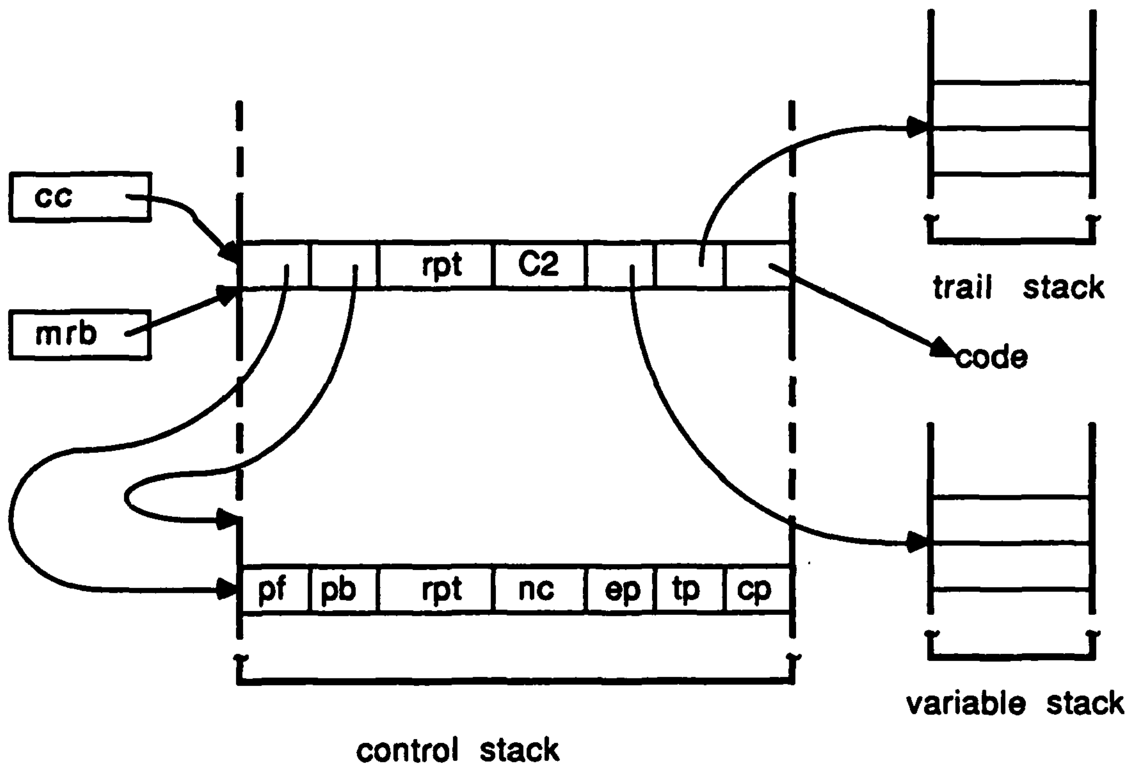


figure VI.2.: The Control Stack

recent choice point is stored in a register called the ***most recent backtrack-point register*** or the mrb register. When mrb needs to be updated, its previous value is stored in the *previous backtrack-point* (pb) field of the current frame. Thus the pb field in the different choice points form an ordered chain of backtrack-points with the first one being in mrb.

When failure occurs, control is transferred to the frame, say *f*, pointed to by the mrb. All the frames above *f* are discarded. The mrb is updated with the value of the *previous backtrack-point* field of the frame *f*. The bindings of the variable recorded on the trail stack between the trail-stack top and the tp field of *f* are undone, and the trail-stack top is set to value of the tp field of *f*. The next candidate to be tried is retrieved using the nc field of *f*. Frame *f* is modified to store the information for the new definition and the computation is restarted.

VI.2.3. Solution of *term* = if-then-else

The interpreter executes the if-then-else like a user defined operation application, except that only the first argument, corresponding to the predicate condition, is evaluated in a call-by-value fashion. As we saw earlier, if the predicate condition is an expression it gets flattened out and replaced by a variable. We will refer to this variable introduced as the *predicate variable*. At the time of evaluation of the if-then-else, the predicate variable can be in one of the three valid states: unbound, bound to the constant *true*, or bound to the constant *false*. If the predicate variable is bound to something else, the if-then-else fails.

The cases when the predicate variable is bound to *true or false* are similar. We therefore explain just the *true* case: When the predicate variable is bound to *true*, the then-part is executed. The interpreter treats the then-part as an anonymous operation application. Both the then-part and the else-part have the general form:

expression where *equations*

resembling the body of a operation. Thus, a frame gets pushed on the control stack after the various fields have been set up. This frame is not a choice point, and hence no backtrack information is required to be stored in it. It also does not get allocated any space on the variable stack, because the if-then-else shares all its variables with the operation it appears in. Thus the environment pointer of the frame for the then-part is the same as the environment pointer of its parent. Once the frame has been set up and pushed, execution continues normally.

When the predicate variable is unbound the if-then-else evaluates to more than one values. Thus, an equation

```
z = if x then 10 else if y then 20 else 30.
```

would yield the following 3 solutions.

```
    x = true
    y = _
    z = 10 ;

    x = false
    y = true
    z = 20 ;

    x = false
    y = false
    z = 30 ;

    no solution
```

Implementing the `if-then-else` with unbound predicate variable is more involved. Although the frame for then-part shares the variable space with its parent frame, it has a separate pointer pointing into the trail-stack because only those variable which got bound as a result of the execution of the then-part need be unbound upon failure. Note that, in an `if-then-else` with an unbound predicate variable, the frame for the then-part can *always* be overwritten with that of the else-part. Thus, the execution of an `if-then-else` never requires more than a single frame. The advantages of treating the `if-then-else` evaluation as a pseudo operation-application are:

i. the control mechanism remains simple and uniform;

ii. it is possible to provide full backtracking on the unbound predicate variable with no extra cost, and

iii. because the `if-then-else` is often deterministic the corresponding frame is not a choice-point. This aids last equation optimization (LEO), discussed later.

The mechanism described above can handle arbitrarily nested `if-then-else`.

VI.2.4. Solution of *term* = *primitive*

To solve an equation with a primitive strict operator, the arguments of the operator are first evaluated, the operation is then applied to arguments, and the result is then unified with *term*. If unification fails backtracking takes place. If one of the operands is not bound, the equation is *delayed*. If *term* is a number, or a variable bound to a number, and the operator is arithmetic (`+`, `-`, `*`, `div`) with only one operand unbound, then the interpreter computes the value of its other operand and binds it. Equation delaying and solution of arithmetic equations is discussed below.

VI.2.4.1. Equation Delaying

To implement equation delaying the interpreter maintains a stack called the *equation delay stack*. When an equation is found to have a primitive operator with unbound operands, a pointer to its code is stored in the equation delay stack. The dsp field in the control frame stores the top of the delay stack whenever a choice point is pushed. On backtracking to this choice point, the top of the delay stack is restored to the value in this field.

Every time a success exit is taken, the interpreter tries to solve all equations on the delay stack. If a delayed equation is solved and belongs to a frame lying below the mrb, a pointer to this equation is also recorded on the *equation trail stack*. This information is recorded so that upon backtracking this equation can be marked as still unsolved, and re-solved later with correct bindings.

VI.2.4.2. Solving Arithmetic Equations

Because of flattening, all arithmetic operations occur in equations of the form:

a = b *op* c, or

b *op* c = a

where each a, b, c is either a variable or number, and *op* is +, -, * or div.

When an arithmetic expression with an unbound operand is found, the term on the other side of that equation is examined. If it is a number, or a variable bound to a number, then the value of the unbound operand is computed and stored in the variable cell of that operand on the variable stack.

Note that equations that have only one occurence of an unbound variable can be solved immediately. Equations of the kind x + y = 20, where x and y are unbound, would be delayed using the mechanism described in the previous section. It can be solved as soon as one of the variable gets bound. Solving such equations, rather than delaying them, makes the interpreter more efficient.

VI.3. Optimizations

So far we have described the core of our interpreter. Now we will discuss some optimizations incorporated in our interpreter. These optimizations are rule-indexing, last-equation optimization, and run-time equation re-ordering. These optimizations lead to considerable efficiency in execution space and time as indicated by our experiments.

VI.3.1. Last Equation Optimization

It is not necessary to delay the reclamation of a frame on the control stack until a *success exit* is taken. It is possible to reclaim a frame, subject to certain conditions, at the time its last equation is being solved. The Last Equation Optimization (LEO) is similar to the Last Call Optimization (LCO) in Prolog implementations and the Tail Recursive Optimization (TRO) in Lisp implementations.

Functional language implementation have fewer opportunities to do TRO due to their expression nesting. Consider the second rule in the functional definition of append:

```
append([], y) => y.
append([h|t], y) => [h| append(t, y)].
```

Since cons is at the outermost level, it is not possible to apply TRO to the recursive call on append. On the other hand LCO can be applied to the recursive call to append in the Prolog definition for append given below:

```
append([], Y, Y).
append([H|T], Y, [H|Z) :- append(T, Y, Z).
```

This is an advantage of the relational style of programming over the functional style — more opportunities for LCO exist.

LEO in EqL is as powerful as LCO in Prolog. That is, LEO would be performed during the execution of EqL code where ever LCO would be performed in the execution of the equivalent Prolog code. This is achieved through compile-time as well as run-time equation reordering. We illustrate this point by considering the above rules for append, and the goal

```
append([1, 2], [3, 4]) = ans.
```

During execution, after the second rule has been matched, this goal would be transformed into:

```
append(t1, y1) = z1;
cons(h1, z1) = ans.
```

Note that `z1` appears due to compile time flattening of append. Due to equation reordering, the second equation would be moved before the first which then triggers a LEO. The conditions for an equation to qualify for LEO are:

i. the equation should be the last to be solved in that operation code, and

ii. the called operation should not be a choice point.

Both these conditions are fairly easy to determine at runtime. In principle, the condition above can be further relaxed, and it is possible to do a LEO when the caller is not a choice point and the callee is. However, this requires extra information to be maintained on the control stack and hence we did not incorporate it in our interpreter. Note that it is the separation of control and data which enables the callee frame (which is not a choice point) to overwrite the calling frame even when the former is a backtrack point.

LEO not only saves space but also time. Space is reduced because frames get deleted from the control stack as soon as they are no longer needed. Programs run faster because when the called frame overwrites the calling frame only 50% of the fields need updating; others (such as the parent frame pointer, return pointer, etc.) remain the same and do not require modification. Thus, overwriting a frame is a much cheaper operation than pushing a new one. Later, we give some statistics showing the improvement in speed due to LEO.

VI.3.2. Rule-indexing

In order to preserve variable bindings during an operation application, a frame needs to be set up on the control stack before unifying the actuals and the formals. Very frequently, this unification fails and the time spent in setting up this frame goes wasted.

Rule-indexing avoids this wasted effort, and is similar to Prolog's clause-indexing [WPP77]. The first actual argument of the operation application is unified with the first formal argument of the operation definition before the execution stacks are set up. The unification algorithm used here is only an approximation to the actual unification algorithm. If rule-indexing succeeds, the execution stacks are set up and true unification is performed. If rule-indexing fails, the next definition is considered in a similar fashion. Even though rule-indexing performs only an approximate unification, our experiments show that it cuts down considerably on the number of potential candidates for an operation application. Since it is performed before the frame for the operation is pushed, it leads to LEO being applied more often, thus multiplying the savings in time and space.

The *approximate* unification algorithm used for rule-indexing is as follows:

i. A constant term unifies with a constant term if they are identical.

ii. An unbound variable unifies with anything.

iii. Two structures unify only if their first components unify (in the *approximate* sense).

iv. An empty list does not unify with a cons structure.

For example, `[x, y]`, if present as the first formal argument would unify with `[1, 2, 3]` in the *approximate* sense, though it wouldn't in the *true* sense.

VI.3.3. Runtime Equation Re-ordering

Run-time equation re-ordering is needed because the new equation that is created during operation application could relate two terms. To illustrate, consider a rule

`f(...) => `*expression* `where` *equations*

and a query equation

`f(...) = `*term*.

Assuming the actuals and formals of f unify, at runtime this equation reduces via the above rule to:

equations;
expression = *term*.

Since the equation *expression* = *term* is generated at run-time, re-ordering is necessary if *expression* turns out to be a term. Runtime equation reordering leads to a greater number of LEOs, and also helps avoid non-termination in some cases. To illustrate this point, consider the query

```
append(x, y) = [1, 2].
```

This goal would yield three solutions. Without run-time equation re-ordering, however, non-termination occurs after the last solution is reported. This non-terminating computation is caused due to the generation of the equations:

```
append(_, _) = _.
[] = cons(..., ...)
```

where _ denotes an unbound variable.

As a result of runtime re-ordering, the equation `[] = cons(..., ...)` is solved first, thus causing termination with failure. The interpreter then reports to the user that no more solutions exist. The reordering also forces the recursive call to append in the second rule to be in the last equation, enabling LEO. As a result of LEO, the entire query gets solved in a *single* frame.

VI.3.4. Experimental Results

The effect of last-equation optimization and rule-indexing may be illustrated from the following typical test cases.

List append

```
append([], x) => x.
append([h|t], y) => [h | append(t,y)].
```

Naive reverse

```
reverse([])    => [].
reverse([h|t])=> append(reverse(t), [h]).
```

Association lists

```
assoc([], x) => 'notfound'.
assoc([[x|v]|t], x) => [x | v].
assoc([[y|v]|t], x) => assoc(t, x) where eq(x, y) = false.
```

The table below summarizes the run-times for these programs. The numbers in the table stand for SUN-3 cpu seconds. The last two digits after the name in the first column of the table indicate the length of the list on which the program was run. The column labeled BASIC lists the timings for the version of the interpreter with no optimizations, the one labeled RI for the version with rule-indexing, and the last for the version with rule-indexing and LEO.

Program	BASIC	RI	RI+LEO
app30	0.045	0.030	0.010
app60	0.100	0.070	0.030
rev30	0.700	0.550	0.410
rev60	2.840	2.300	1.610
assoc30	0.070	0.065	0.060

The improvements from the optimizations should be obvious from the table. We observed similar improvements with other programs as well. Note that the time taken to traverse 30 elements of an association list is considerably more than the time taken to append a 30-element list to another list. This supports the observation that structure sharing favors data construction over data access [M80]. (Note that printing time for the resulting lists is not included in the above timings.)

VII. Conclusions

Although EqL supports functional programming more directly than logic programming (because of its functional syntax), we hope it is clear from the examples that many logic programming paradigms are also readily supported. In fact any pure Prolog program can be directly converted into an EqL program [JS86]. EqL programs are often clearer than equivalent Prolog programs because function composition supplants the need for "output variables" in Prolog programs (see definition of *naive* reverse, for example). EqL's if-then-else corresponds to a well-structured use of Prolog's cut, and its use also leads to simpler formulations in many cases. EqL supports arithmetic, relational and boolean primitives, and delays any equation with these primitives until sufficient information becomes available. Other useful features in the language are strings and user-defined constructors.

The development of EqL was inspired by several recents efforts: O'Donnell first introduced the term 'equational programming' to refer to a style of functional programming with equations [HO82, O85]. You and Subrahmanyam extended this language for logic programming [YS86] by permitting equations to be solved only at the top-level goal. EqL further extends this approach and permits equations at the top-level as well as at intermediate levels. Note that EqL rules are more general in that variables on the right-side of a rule need not appear on the left-side—this generality is, in fact, needed to attain the full expressiveness of Horn logic.

The operational strategy of EqL is related to *narrowing* in term-rewriting systems. Unrestricted narrowing has excessive computational requirements because many reductions and narrowings are possible at each step, and these are not readily located—the complexity of a narrowing implementation can be seen from [JD86]. *Lazy narrowing* restricts the matching to the outermost term, but, unlike ordinary lazy evaluation [HM76], argument expressions may have to be re-narrowed for each different rule for an operation [R85]—because the extent of (lazy) narrowing could depend on the left-hand side of a rule. In contrast, because of its call-by-value semantics, EqL avoids such re-evaluation; EqL's equation-solving procedure may be viewed as a form of *innermost narrowing*.

The EqL interpreter divides into two phases: compilation and interpretation. Compilation involves generation of binary-tree intermediate code, accompanied by flattening and equation re-ordering transformations. The interpretation phase solves the different forms of equations using five stacks, each with fixed size entries. From our experiments, we found that last equation optimization and rule indexing improve runtime performance considerably. Note that because of the complete separation of the control and variable stacks, the variable stack shrinks only during backtracking. Garbage collection is therefore needed when no more space is left on it. Our current implementation does not support garbage collection yet.

For the sake of storage efficiency, our sequential implementation searches for solutions depth-first with backtracking, similar to Prolog implementations. Although depth-first search does sacrifice *completeness* in theory, we have not found this to be a serious limitation in practice. Ultimately performance for such a language must be realized through parallelism—which would also facilitate a *complete* implementation. Because sequential machines are still in widespread use, we felt it was worthwhile investigating efficient sequential implementations. The availability of a sequential implementation has been extremely valuable to us in experimenting with this new programming paradigm.

We completed Version 1.0 of the interpreter in May 1987, and have successfully run several non-trivial programs using it. The EqL interpreter runs at about half the speed of the C-Prolog interpreter for standard benchmark programs such as naive reverse. By compiling EqL programs to WAM-like code [W83], comparable performance with Prolog can be expected. We nevertheless hope we have demonstrated that the elegance of functional notation (composition, if-then-else) and the expressiveness of logic (non-determinism, ability to declare constraints) may be combined (via equation solving) to yield a practical declarative language.

References

[BM72] R. S. Boyer and J. S. Moore, "The sharing of structure in theorem proving programs," In *Machine Intelligence* 7, B. Meltzer and D. Michie (eds.), 1972, pp. 101-116.

[CM81] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.

[DP85] N. Dershowitz and D. A. Plaisted, "Applicative Programming *cum* Logic Programming," In *1985 Symp. on Logic Programming*, Boston, 1985, pp. 54–66.

[F85] L. Fribourg, "SLOG: A Logic Programming Language Interpreter based on Clausal Superposition and Rewriting," In *1985 Symp. on Logic Programming*, pp. 172-184, Boston, 1985.

[GM84] J. A. Goguen and J. Meseguer, "Equality, Types, Modules, and (Why Not?) Generics for Logic Programming," *J. Logic Prog.* **2** (1984) pp. 179–210.

[H84] C.J. Hogger, "Introduction to Logic Programmming," Academic Press, 1984.

[HHT82] A. Hansson, S. Haridi, and S.-A. Tärnlund, "Properties of a Logic Programming Language," In *Logic Programming*, Ed. K. L. Clark and S.-A. Tärnlund, Academic Press, 1982, pp. 267–280.

[HM76] P. Henderson and J. H. Morris, "A Lazy Evaluator." In *Third ACM POPL*, 1976, pp. 95–103.

[HO82] C. M. Hoffman and M. J. O'Donnell, "Programming with Equations," *ACM TOPLAS* 4, No. 1, pp. 83–112, January 1982.

[JD86] A. Josephson and N. Dershowitz, "An Implementation of Narrowing: the RITE Way," In *IEEE Symp. on Logic Prog.*, Salt Lake City, September 1986, pp. 187-197.

[J87] B. Jayaraman, "Semantics of EqL," To appear in *IEEE Trans. on Software Engg.*, 1987.

[JG87] B. Jayaraman and G. Gupta, "EqL User's Guide," TR 87-010, Dept. of Comp. Science, UNC - Chapel Hill, May 1987, 30 pages.

[JL87] J. Jaffar, J.-L. Lassez, "Constraint Logic Programming," In *14th ACM POPL*, pp. 111-119, Munich, West Germany, 1987.

[JS86] B. Jayaraman and F.S.K. Silbermann, "Equations, Sets, and Reduction Semantics for Functional and Logic Programming," *In 1986 ACM Symposium on LISP and Functional Programming*, pp. 320-331, Boston, 1986.

[L85] G. Lindstrom, "Functional Programming and the Logical Variable," In *12th ACM POPL*, New Orleans, Jan 1985, pp. 266–280.

[M80] C.S. Mellish, "An alternative to structure-sharing in the implementation of a Prolog interpreter," D.A.I. Research Paper No. 150, Univ. of Edinburgh, 1980.

[O85] M. J. O'Donnell, "Equational Logic as a Programming Language," M.I.T. Press, 1985.

[R85] U. S. Reddy, "Narrowing as the Operational Semantics of Functional Languages," In *1985 Symp. on Logic Programming*, Boston, 1985, pp. 138–151.

[RS82] J. A. Robinson and E. E. Sibert, "LOGLISP: Motivation, Design, and Implementation," In *Logic Programming*, Ed. K. L. Clark and S.-A. Tärnlund, Academic Press, 1982, pp. 299–313.

[W83] D. H. D. Warren, "An Abstract Prolog Instruction Set," Tech. Note 309, SRI International, Menlo Park, October 1983.

[WPP77] D. H. D. Warren, F. Pereira, and L. M. Pereira, "Prolog: the Language and Its Implementation Compared with LISP," *SIGPLAN Notices* **12**, No. 8 (1977) pp. 109–115.

[YS86] J-H. You and P. A. Subrahmanyam, "Equational Logic Programming: an Extension to Equational Programming," In *13th ACM POPL*, St. Petersburg, 1986, pp. 209-218.